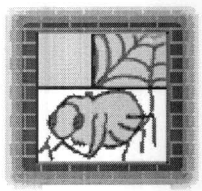
The ACM Student Magazine

Charlotte: A Simple Web Server for Microsoft Windows

by ***Stuart Patterson***

Introduction



The idea of writing a web server first struck me when I was investigating Java in 1996. After reading through the language documentation, I decided that by developing a simple application I could learn the more esoteric aspects of Java -- multithreading and sockets. The outcome of my Java experimentation was Charlotte, a simple web server named after the small and wise spider from the book, *Charlotte's Web*. Since I first created Charlotte, it has spent most of its life tucked away on a floppy disk, a long forgotten victim of my fickle interests and academic demands. Wanting to gain the same educational benefits I experienced with the Java version, I recently began developing a Windows C/C++ version. As a result of the rewrite, Charlotte has been reborn as a 32-bit Windows Console application that utilizes multithreading and the Windows socket library, "Winsock." This article explains how Charlotte was written for the Windows operating system.

Sockets & HTTP 101

Before we examine the code, an introduction to Sockets and the Hypertext Transfer Protocol (HTTP) is required. Computers, connected to the Internet, communicate with one another using the Internet Protocol (IP). The transport layer on top of IP supports two primary methods by which machines can exchange information: a reliable, connection-oriented, Transmission Control Protocol (TCP) and a connectionless, unreliable, User Datagram Protocol (UDP). This article focuses solely on TCP sockets, the communication protocol used by web browsers and servers.

Before an application can call any socket functions, Windows requires the Winsock library to be initialized via a call to `WSAStartup()`. To simplify this article, I do not discuss function parameters. Interested readers should explore function parameters in detail. Once Winsock is initialized, a socket can be created to allow two machines to communicate. The term **socket** describes an end-point of communications through which data can be sent and received. Sockets are created by calling the Winsock function `socket()`. Using a socket differs between client-based and server-based applications. Clients need only call `connect()` to establish a connection to a server. Servers are also required to further prepare the socket for incoming client connections. **Figure 1** shows a flow diagram describing the function calls needed for both a server and a client TCP/IP application. Since we are developing a simple web server, we will focus on server-based sockets.

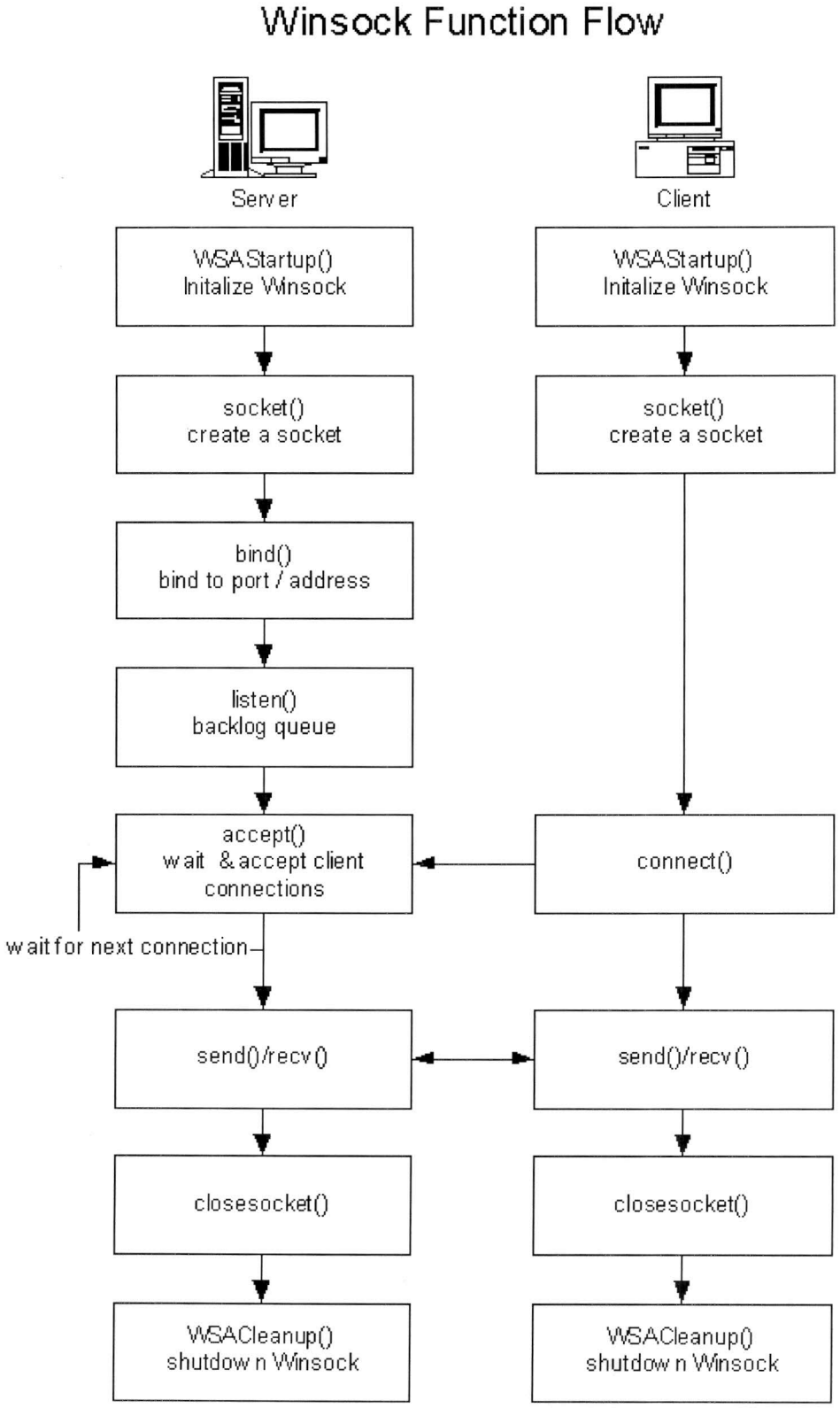


Figure 1 - Function Flow

Once a socket is created, a server needs to bind the socket using the `bind()` function. Binding a socket ties it to a given port and IP address. By providing services on different ports, a machine can host a number of different applications. For instance, it is common to host both FTP and HTTP on the same machine. Typically, HTTP uses port 80 for all incoming client connections; Charlotte is currently hard-coded to port 80. Once the port is bound to the socket, the server calls `listen()` to specify the maximum length of the socket's backlog queue. Since a server must handle simultaneous client connections, the backlog queue allows the server to hold pending client connections until the server can accept them. In Charlotte, this value has been set to `SOMAXCONN`, a manifest constant defined in the Winsock header, which forces the TCP/IP stack provider to set the maximum backlog value for the socket. The socket is now ready to accept incoming client browser connections. Calling the `accept()` function causes the server to wait until a client connection is detected, at which point `accept()` returns with a new socket specifically for communicating with that client. Using the returned socket and `recv()`, we can read the HTTP request that the client sent and send back the proper document (HTML, image, etc.) using `send()`. When an application is through using a socket, the socket must be closed by calling `closesocket()`. Before the application exits, it must release the Winsock resources via a call to `WSACleanup()`.

As you will see in the accompanying source code, some of the TCP/IP functions expect data to be transferred in network byte order (big-endian). Within the Winsock library are a number of functions to aid in the conversion between Intel byte order (little-endian) and network byte order. **Little-endian** means that the least significant byte of a word comes first, whereas **big-endian** means the opposite.

The World Wide Web is based on the Hypertext Transfer Protocol (HTTP). This protocol defines how a client browser will request a document from a web server once it has connected via sockets. It might be easier to think of a socket as a telephone (the low-level communication device) and HTTP as the messages conveyed over the phone. The following diagram should clarify the protocol layers used in an HTTP request:

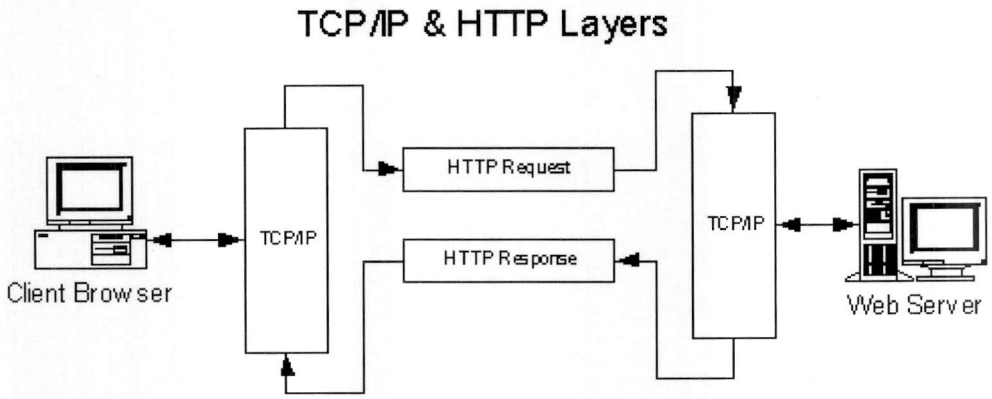


Figure 2 - TCP/IP & HTTP Layers

Currently, Charlotte only supports the bare minimum requirements of a web server as defined in RFC1945 HTTP version 1.0. A client browser connects to a web server using TCP/IP on port 80. The client browser sends a request in the form of "a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content" [3]. Multipurpose Internet Mail Extension (MIME) is defined in RFC1521 and RFC1522 and is used by HTTP to identify the data being sent to the client browser. A URI, or Uniform Resource Identifier, describes the location of a document, such as http://www.mysite.com/article.html. The URI sent from the client browser to the web server in the request header is a shorter form describing the document relative to the server, such as /article.html. HTTP 1.0 supports four different request methods, GET, POST, HEAD, and method extensions. The GET request is the most common and the only method currently supported by Charlotte. A GET request informs the server that the client browser would like to retrieve a copy of the document described in the Request-URI. A client browser sends a request in clear ASCII text with the first line containing the method, URI, and protocol version. Here is a typical browser request made from my web browser to Charlotte:

```
GET / HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/msword, application/vnd.ms-excel, application/x-comet,
application/vnd.ms-powerpoint, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows NT; DigExt)
```

```
Host: localhost
Connection: Keep-Alive
```

The last line of an HTTP request is a carriage-return/line-feed pair without any other data on the line. This allows Charlotte to determine when the client has sent a complete HTTP request. By parsing the first line we can determine what document the client is requesting from our server. Then, using the C run-time library file functions, we can retrieve the file and send it back to the client. The only information Charlotte uses is the first line of the request. The other lines allow more advanced web servers to make programmatic decisions based on the document type and browser versions. Charlotte uses the Request-URI to determine the location of the file in relation to the root path of the web server. The web server's root path is set when Charlotte is started from the command-line:

```
c:\>charlotte c:\wwwroot
```

The web server then takes the Request-URI (in this case '/'), appends it to the root web directory, and retrieves the file. In the case of '/', Charlotte can determine that the user is requesting the default html document, index.html, since '/' does not reference a valid file or subdirectory. Allowing the client to request a directory and view it's file listing is not defined as part of the HTTP specification, but is implemented in commercial web servers by interpreting the default document request '/' as an inquiry for the directory contents.

Code Walk Through

Trying to keep a code walkthrough interesting can be a difficult task. I do not want to overlook important ideas within the code, but I also do not want to bore the reader with easily understood concepts. With this in mind, I have decided to skip concepts explained in the sockets and HTTP overviews, focusing on the more complex or unique code. I did not include all the source code in my walkthrough, since much of it is boilerplate in nature. All **source code** and **make files** are included with the article at the ACM Crossroads website.

```
//  Initializes winsock and starts server

int main(int argc, char *argv[]) {
```

```
WSADATA wd;
SOCKET server_socket;

// webserver root directory passed on command-line
if ( argc == 1 ) {
   cout << "charlotte <wwwroot directory>" << endl;
   cout << "ex: charlotte c:\\wwwroot" << endl;
   return(1);
}
// remember the root dir of the webserver
strcpy(wwwroot,argv[1]);
cout << "wwwroot: " << wwwroot << endl;
// init the winsock libraries
if ( WSAStartup(MAKEWORD(1,1), &wd) != 0 ) {
   cout << "Unable to initialize WinSock 1.1" << endl;
   return(1);
}
// get name of webserver machine. needed for redirects
// gethostname does not return a fully qualified host name
DetermineHost(hostname);
```

`DetermineHost()` gets the fully qualified host name of the current machine that Charlotte is running on. The machine name is needed so that when a client requests a document such as '/' the server can redirect the client to the wwwroot/index.html file. Redirects are a common method of forwarding the client browser to the proper default document.

```
// create a critical section for
// mutual-exclusion synchronization
// on cout
InitializeCriticalSection (&output_criticalsection);
```

Critical Sections are used to limit thread access to a shared resource. In this case the Critical Section protects writing to the I/O stream cout. If some form of mutual-exclusion object were not used and the threads called cout at the same time, each thread could overwrite the other's output.

```
// init the webserver
```

```
    server_socket = StartWebServer();
```

`StartWebServer()` creates the socket and initializes it to a port and address via a call to `bind()`. If the socket is successfully created, a call to `WaitForClientConnections()` is made, passing the server_socket. `WaitForClientConnections()` then calls `listen()` and `accept()` on the server_socket. `WaitForClientConnections()` does not return. It spins in an infinite loop, waiting on each client connection. To terminate the web server you can type Ctrl-C at the console window.

```
    if ( server_socket ) {
        WaitForClientConnections(server_socket);
        closesocket(server_socket);
    }
    else cout << "Error in StartWebServer()" << endl;
    // delete and release resources for critical section
    DeleteCriticalSection (&output_criticalsection);
    WSACleanup();
    return(0);
}

// Loops forever waiting for client connections. On connection
// starts a thread to handling the http transaction
int WaitForClientConnections(SOCKET server_socket) {
    SOCKET client_socket;
    SOCKADDR_IN client_address;
    int client_address_len;
    ClientInfo *ci;

    client_address_len = sizeof(SOCKADDR_IN);
    if ( listen(server_socket,SOMAXCONN) == SOCKET_ERROR ) {
        OutputScreenError("Error in listen()");
        closesocket(server_socket);
        return(0);
    }
    // loop forever accepting client connections. user ctrl-c to exit!
    for ( ;; ) {
        client_socket = accept(server_socket,
```

```
                (struct sockaddr*)&client_address,
                &client_address_len);
```

The `accept()` function blocks, waiting for incoming client connections. When a connection arrives, accept() returns the client socket and the IP address information of the client.

```
      if ( client_socket == INVALID_SOCKET ) {
         OutputScreenError("Error in accept()");
         closesocket(server_socket);
         return(0);
      }
      // copy client ip and socket so the HandleHTTPRequest thread
      // and process the request.
      ci = new ClientInfo;
      ci-> client_socket = client_socket;
      memcpy(&(ci->client_ip),
                &client_address.sin_addr.s_addr,4);
      // for each request start a new thread!
      _beginthread(HandleHTTPRequest,0,(void *)ci);
   }
}
```

Each client request is handled in its own thread, thus allowing the web server to asynchronously manage incoming connections in a responsive manner. Client data is sent to the thread by first placing it into a ClientInfo structure. An instance of the structure is created for each thread. This protects the data from thread overwrites and allows the thread to be autonomous. Threads are created via `_beginthread()`, which passes a ClientInfo pointer to `HandleHTTPRequest()`.

Running in its own thread, `HandleHTTPRequest()` handles each individual client request.

```
// Executed in its own thread to handling http transaction
void HandleHTTPRequest( void *data ) {
   SOCKET client_socket;
   HTTPRequestHeader requestheader;
   int size;
```

```
char receivebuffer[COMM_BUFFER_SIZE];
char sendbuffer[COMM_BUFFER_SIZE];

client_socket =
     ((ClientInfo *)data)->client_socket;
requestheader.client_ip =
     ((ClientInfo *)data)->client_ip;
delete data;
size = SocketRead(client_socket,
     receivebuffer,COMM_BUFFER_SIZE);
```

`SocketRead()` is a `recv()` function wrapper. The `recv()` function blocks until the client has sent some of its data to the server. SocketRead() continues reading and buffering client data until it detects a single line containing only a carriage return/line feed, indicating the end of the client HTTP request.

```
if ( size == SOCKET_ERROR || size == 0 ) {
   OutputScreenError("Error calling recv()");
   closesocket(client_socket);
   return;
}
receivebuffer[size] = NULL;
if ( !ParseHTTPHeader(receivebuffer,requestheader) ) {
   // handle bad header!
   OutputHTTPError(client_socket, 400);   // 400 - bad request
   return;
}
```

`ParseHTTPHeader()` takes the receivebuffer returned from `SocketRead()` and parses out the first line, placing it into a HTTPRequestHeader structure.

```
if ( strstr(requestheader.method,"GET") ) {
```

Check to verify that the request method is a GET.

```
   if ( strnicmp(requestheader.filepathname,
            wwwroot,strlen(wwwroot))== 0 )
            // else security violation!
   {
```

Next, compare the requestheader.filepathname that was calculated in `ParseHTTPHeader()` function with the wwwroot path. If the file being requested is in the wwwroot path, Charlotte can safely retrieve the document. Otherwise, a security error is returned to the client.

```
FILE *in;
char *filebuffer;
long filesize;
DWORD fileattrib;

fileattrib =
       GetFileAttributes(requestheader.filepathname);
```

`GetFileAttributes()` is called to determine if the file requested exists or if the file represents a subdirectory. If the user requested http://my.machine.com/, then requestheader.filepathname will contain the wwwroot path. Thus, `GetFileAttributes()` will return with the FILE_ATTRIBUTE_DIRECTORY bit set, and Charlotte will need to redirect the client to the default HTML file via a call to `OutputHTTPRedirect()`.

```
if (fileattrib != -1 &&
       fileattrib &
       FILE_ATTRIBUTE_DIRECTORY) {
   OutputHTTPRedirect(client_socket,
                requestheader.url);
   return;
}
in = fopen(requestheader.filepathname,"rb");  // read binary
if ( !in ) {
   // file error, not found?
   OutputHTTPError(client_socket, 404);    // 404 - not found
   return;
}
```

If the requestheader.filepathname is not a directory, Charlotte tries to open the file. If opening the file fails, an error message is sent to the client.

```
// determine file size
fseek(in,0,SEEK_END);
```

```
filesize = ftell(in);
fseek(in,0,SEEK_SET);

// allocate buffer and read in file contents
filebuffer = new char[filesize];
fread(filebuffer,sizeof(char),filesize,in);
fclose(in);
```

If the file exists, it is opened and its size is determined. A buffer large enough to hold the file contents is allocated, and the file is copied into the buffer. When the copy is complete, the file is closed.

```
// send the http header and the file contents to the browser
strcpy(sendbuffer,"HTTP/1.0 200 OK\r\n");
strncat(sendbuffer,"Content-Type: ",COMM_BUFFER_SIZE);
strncat(sendbuffer,
        mimetypes[findMimeType(requestheader.filepathname)].mime,
        COMM_BUFFER_SIZE);
sprintf(sendbuffer+strlen(sendbuffer),
        "\r\nContent-Length:%ld\r\n",filesize);
strncat(sendbuffer,"\r\n",COMM_BUFFER_SIZE);
send(client_socket,sendbuffer,strlen(sendbuffer),0);
send(client_socket,filebuffer,filesize,0);
```

With the filebuffer loaded with the file contents, Charlotte then builds an HTTP response in the sendbuffer. The response tells the client that its request is OK and lists the proper MIME type and size of the accompanying document. MIME types allow the browser to identify the type of document being returned. Charlotte uses the global array, mimetypes[], to determine, based on file extension, what the MIME type should be for each requested document. The sendbuffer and the filebuffer are both sent to the client via the `send()` function call.

```
// log line
EnterCriticalSection (&output_criticalsection);
cout << inet_ntoa(requestheader.client_ip)
         << " - " << requestheader.method << " "
         << requestheader.url << endl;
LeaveCriticalSection (&output_criticalsection);
```

Next, the client IP address and client request are sent to stdout by first calling `EnterCritcalSection()`, which synchronizes access to the pending cout. `LeaveCriticalSection()` is then called to release the Critical Section, allowing other threads to execute the cout statement.

```
            delete [] filebuffer;
        }
        else {
            OutputHTTPError(client_socket, 403);    // 403 - forbidden
            return;
        }
    }
    else {
        OutputHTTPError(client_socket, 501);    // 501 not implemented
        return;
    }
```

Finally, the client socket is closed.

```
    closesocket(client_socket);
}
```

That explains the basic functionality of Charlotte. By using a separate thread for each client request, blocking is eliminated between each client connection. Thread synchronization using a Critical Section allows each thread to write its output to stdout without the concern of corrupting the output. I tested Charlotte on both single and dual processor machines running Windows NT 4.0 Workstation and Server. On a dual processor Pentium II 350 MHz machine running Windows NT 4.0 Server with 256 Megs of RAM, I successfully handled 17,845 requests in 2:01 minutes, averaging 147 requests per second.

Building

Charlotte was developed in Visual C++ 6.0 and uses the multithreaded version of the C run-time library as well as the Winsock library ws2_32.lib. Except for a few Win32 SDK calls, the source code should be easy to port to any operating system compatible with Berkeley Sockets.

Conclusion

Developing Charlotte gave me the opportunity to experiment with sockets and threads under the Windows operating system. There are numerous opportunities to further expand and enhance this small web server, adding better security, error handling, support for POST request methods, decoding URL encoded requests, and interfacing with CGI applications. I hope this article has introduced you to the excitement of TCP/IP based client/server application development.

References

1

Dumas, Arthur. *Programming Winsock.* Sams Publishing, Indianapolis, Indiana. 1995.

2

Davis, Ralph. *Win32 Network Programming.* Addison Wesley Developers Press, Reading Massachusetts. 1996.

3

Berners-Lee, T and Fielding, R and Frystyk, H. *Rfc1945 Hypertext Transfer Protocol -- HTTP/1.0* **http://www.cis.ohio-state.edu/htbin/rfc/rfc1945.html**,May 1996

4

Microsoft. *Microsoft Developers Network (MSDN)* April 1999

Published Articles:

"A GotoUrl function using ShellExecute()" , August 1997, Windows Developers Journal